

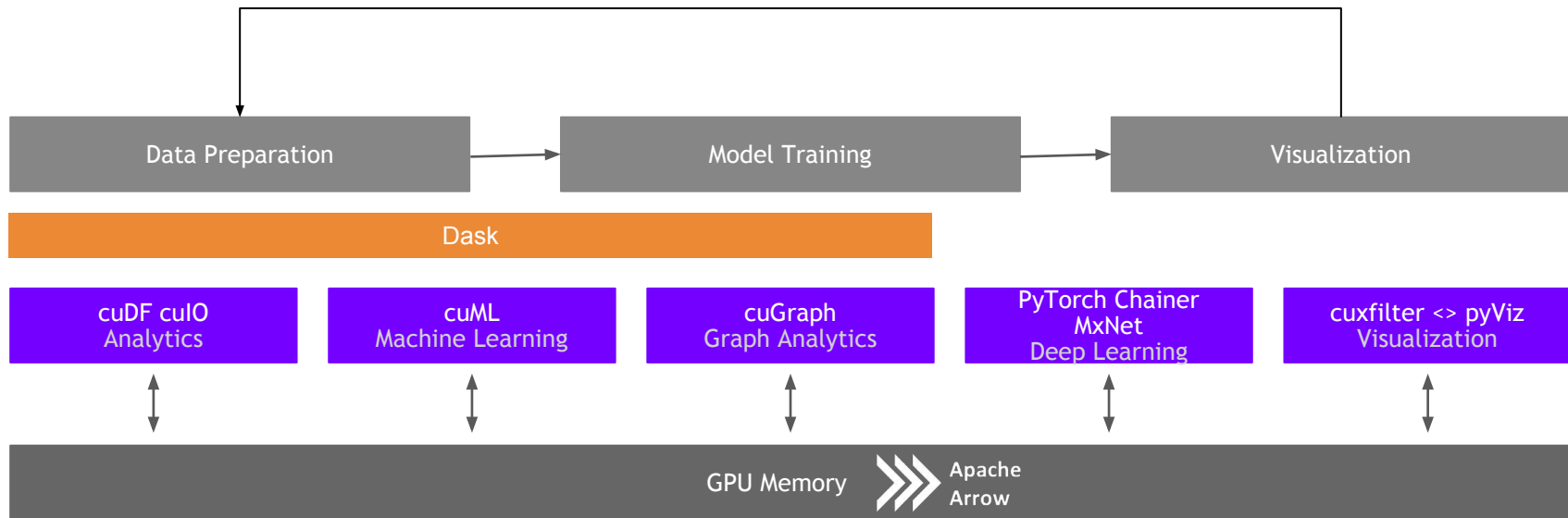
# RAPIDS

## The Platform Inside and Out

Nick Becker  
RAPIDS Engineering

# RAPIDS

## End-to-End Accelerated GPU Data Science



# Data Processing Evolution

Faster data access, less data movement

Hadoop Processing, Reading from disk



Spark In-Memory Processing

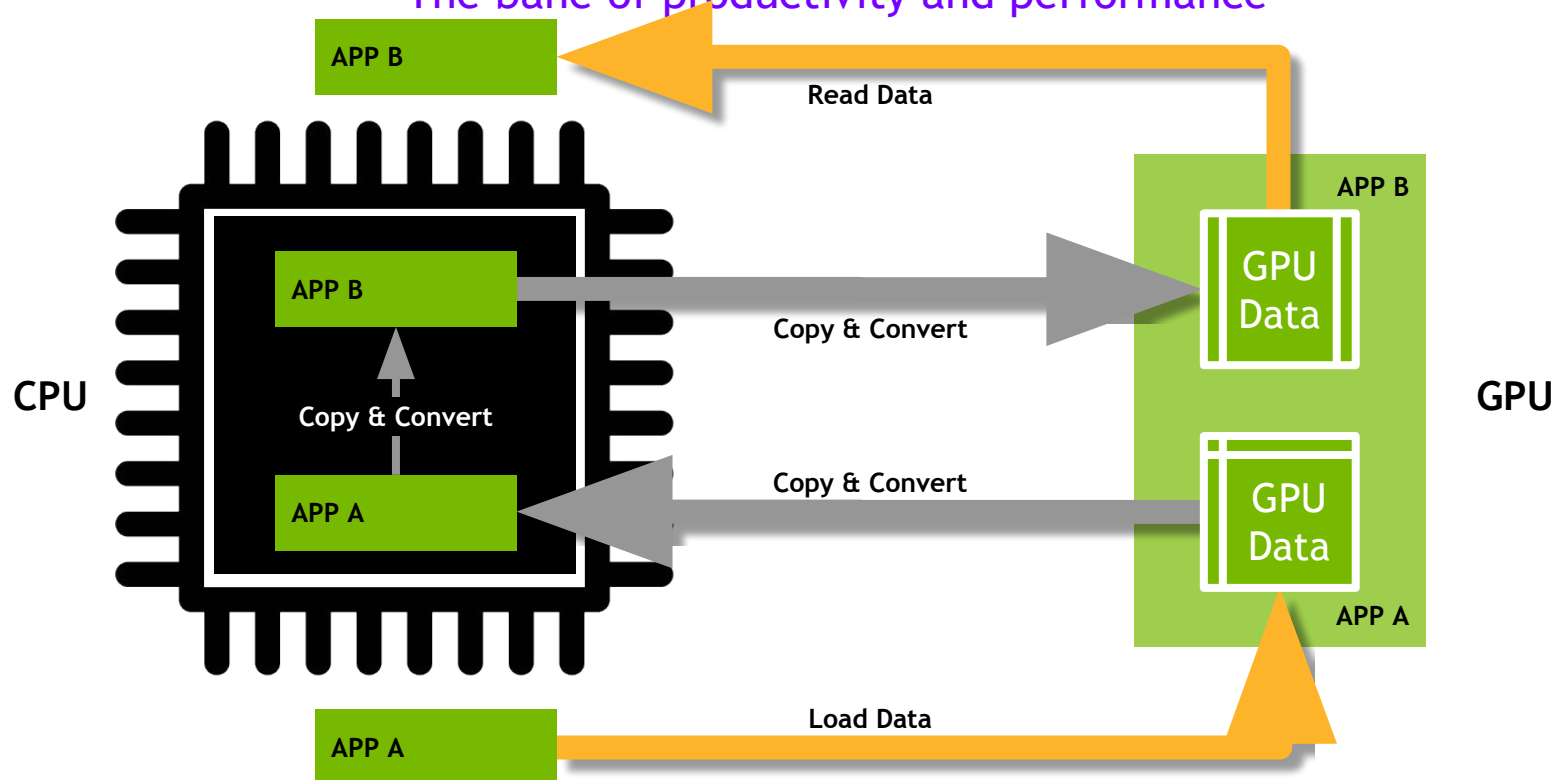


Traditional GPU Processing



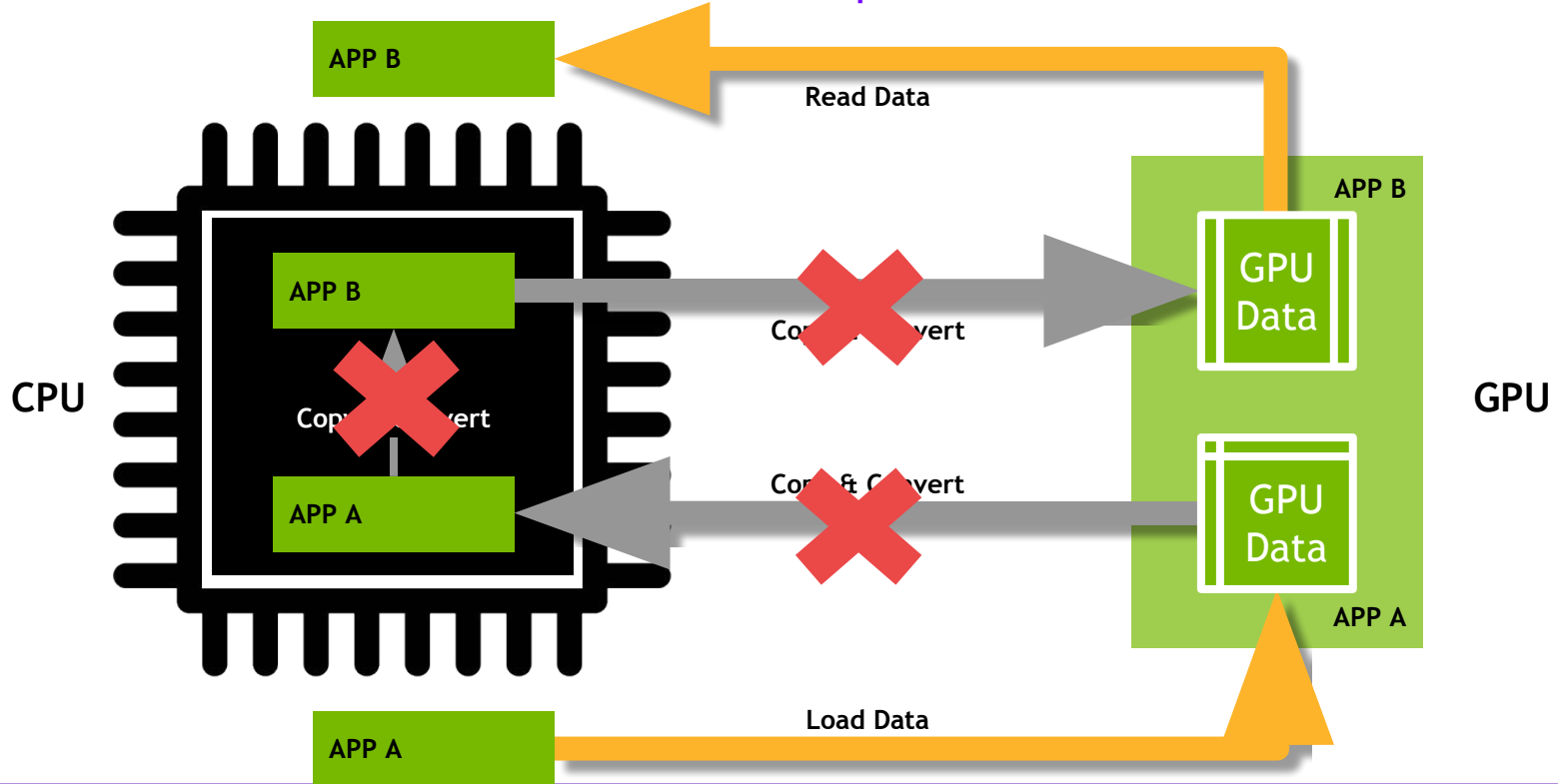
# Data Movement and Transformation

The bane of productivity and performance

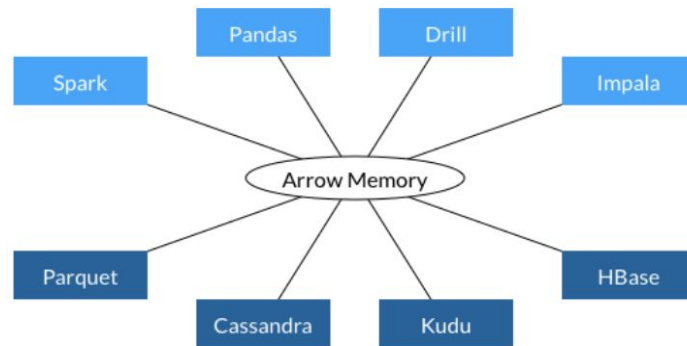
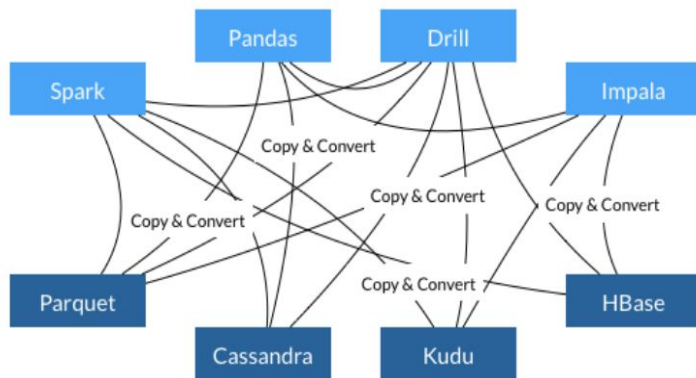


# Data Movement and Transformation

What if we could keep data on the GPU?



# Learning from Apache Arrow >>>



- Each system has its own internal memory format
- 70-80% computation wasted on serialization and deserialization
- Similar functionality implemented in multiple projects
- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (eg, Parquet-to-Arrow reader)

From Apache Arrow Home Page - <https://arrow.apache.org/>

# Data Processing Evolution

Faster data access, less data movement

Hadoop Processing, Reading from disk



Spark In-Memory Processing



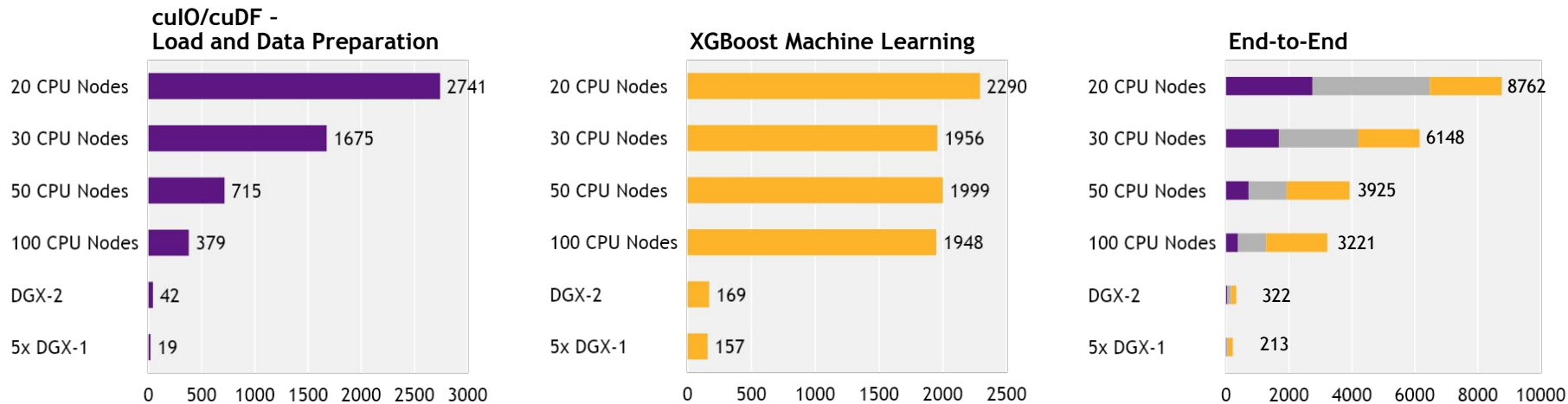
Traditional GPU Processing



RAPIDS



# Faster Speeds, Real-World Benefits



Time in seconds (shorter is better)

■ cuIO/cuDF (Load and Data Prep) ■ Data Conversion ■ XGBoost

## Benchmark

200GB CSV dataset; Data prep includes joins, variable transformations

## CPU Cluster Configuration

CPU nodes (61 GiB memory, 8 vCPUs, 64-bit platform), Apache Spark

## DGX Cluster Configuration

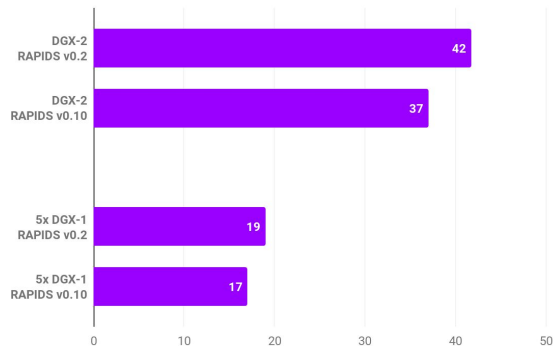
5x DGX-1 on InfiniBand network



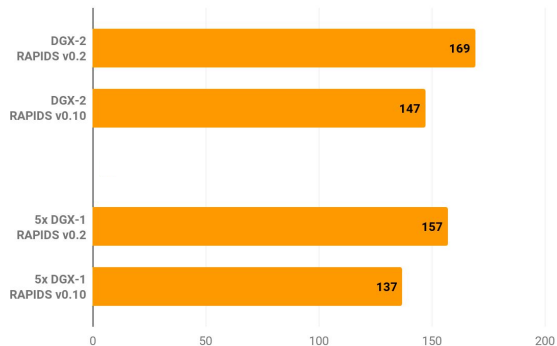
# Faster Speeds, Real-World Benefits

## Improving Over Time

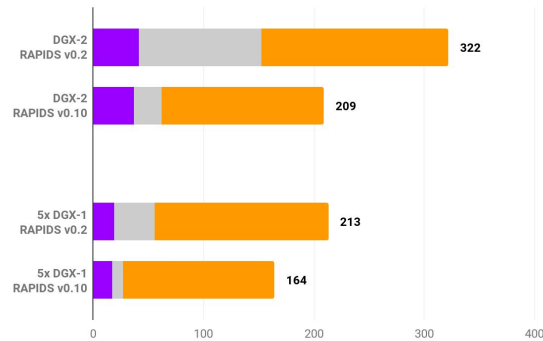
cuIO/cuDF -  
Load and Data Preparation



XGBoost Machine Learning



End-to-End



Time in seconds (shorter is better)

■ cuIO/cuDF (Load and Data Prep) ■ Data Conversion ■ XGBoost

### Benchmark

200GB CSV dataset; Data prep includes joins, variable transformations

### CPU Cluster Configuration

CPU nodes (61 GiB memory, 8 vCPUs, 64-bit platform), Apache Spark

### DGX Cluster Configuration

5x DGX-1 on InfiniBand network

# Speed, Ease of Use, and Iteration

## The Way to Win at Data Science

**François Chollet** @fchollet · Apr 3  
Following

Winners are those who went through "more iterations" of the "loop of progress" -- going from an idea, to its implementation, to actionable results. So the winning teams are simply those able to run through this loop "faster".

And this is where Keras gives you an edge.

12:31 PM - 3 Apr 2019

50 Retweets 158 Likes

**François Chollet** @fchollet · Apr 3  
We often talk about how following UX best practices for API design makes Keras more accessible and easier to use, and how this helps beginners.

But those who stand to benefit most from good UX "aren't" the beginners. It's actually the very best practitioners in the world.

**François Chollet** @fchollet · Apr 3  
Because good UX reduces the overhead (development overhead & cognitive overhead) to setting up new experiments. It means you will be able to iterate faster. You will be able to try more ideas.

And ultimately, that's how you win competitions or get papers published.

**François Chollet** @fchollet · Apr 3  
So I don't think it's mere personal preference if Kaggle champions are overwhelmingly using Keras.

Using Keras means you're more likely to win, and inversely, those who practice the sort of fast experimentation strategy that sets them up to win are more likely to prefer Keras.

**Joshua Patterson** @datametrician · Apr 3  
Replying to @fchollet  
This is the fundamental belief that drives @RAPIDSai. @nvidia #GPU infrastructure is fast, people need to iterate quickly, people want a known #python interface. Combine them and you're off to the races!

**François Chollet** @fchollet · Apr 3  
The second question asked about secondary frameworks -- usually teams win with an ensemble that involves many different ML frameworks. Here are "all" frameworks used.

Sklearn tops that ranking: everyone uses sklearn (although often as an auxiliary, for preprocessing or scoring).

Framework	Category	Count (approx.)
Scikit-Learn	Classic	80
Keras	Deep	65
LightGBM	Classic	55
XGBoost	Classic	55
PyTorch	Deep	35
TensorFlow (non-Keras)	Deep	25
Caffe	Deep	10
MXNet	Deep	10
Fastai	Deep	10
Caffe2	Deep	10
CatBoost	Classic	10
R Random Forest	Classic	10

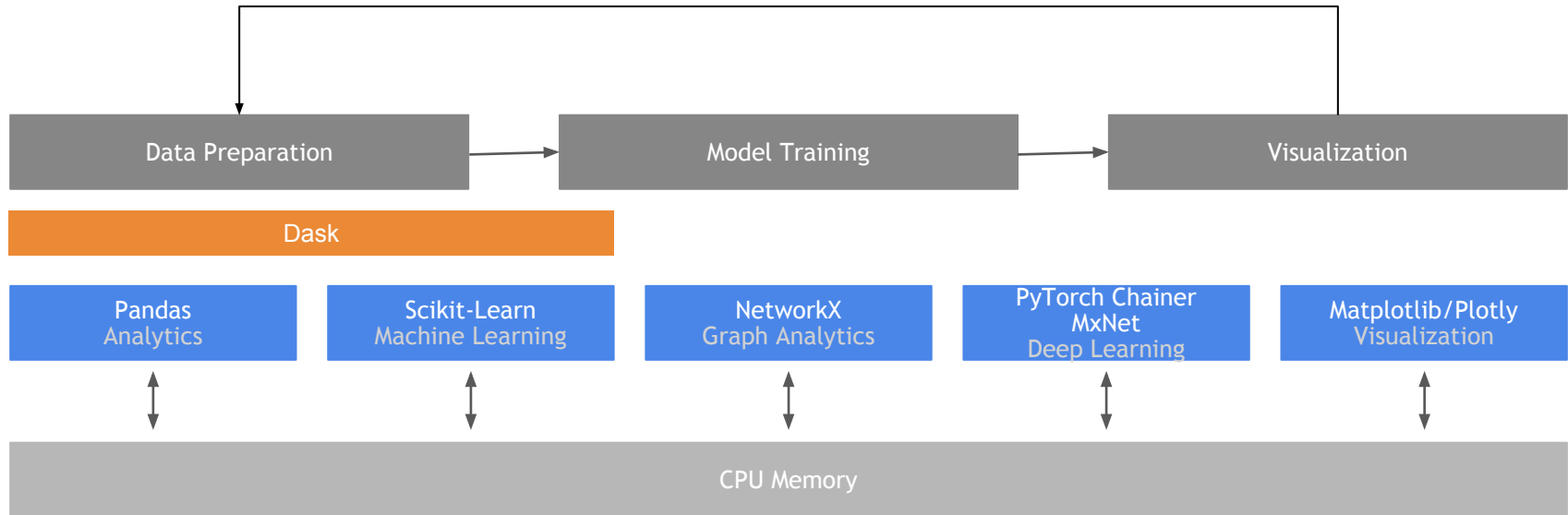
4 44 129

kaggle

# RAPIDS Core

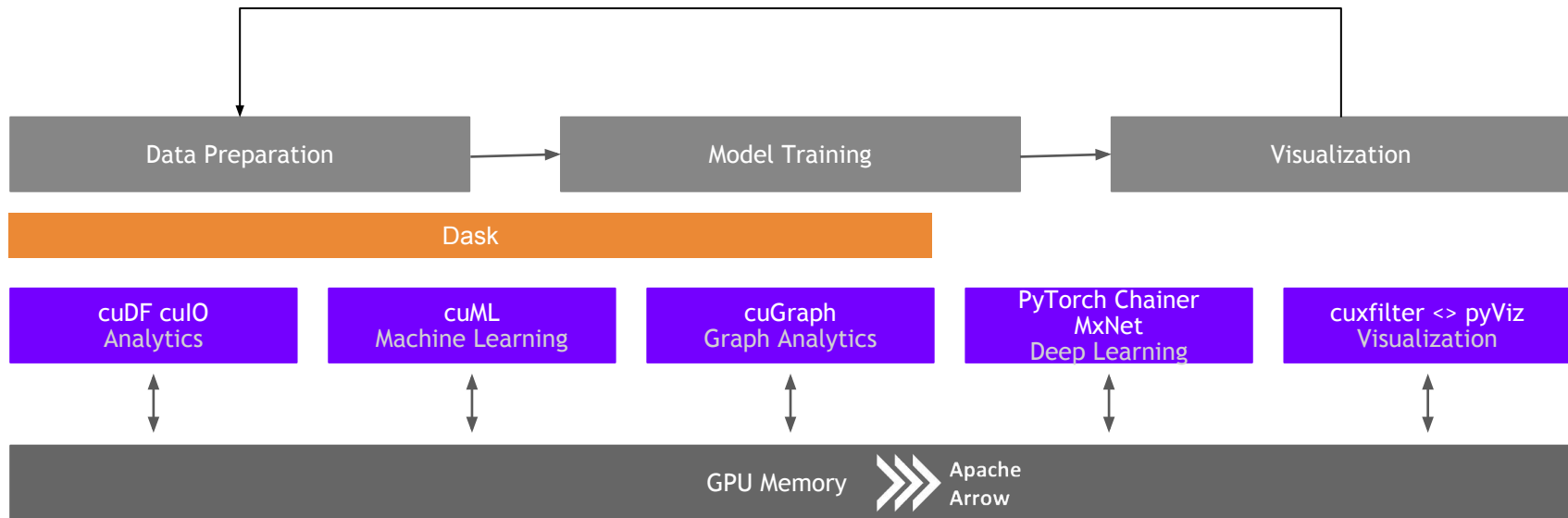
# Open Source Data Science Ecosystem

## Familiar Python APIs



# RAPIDS

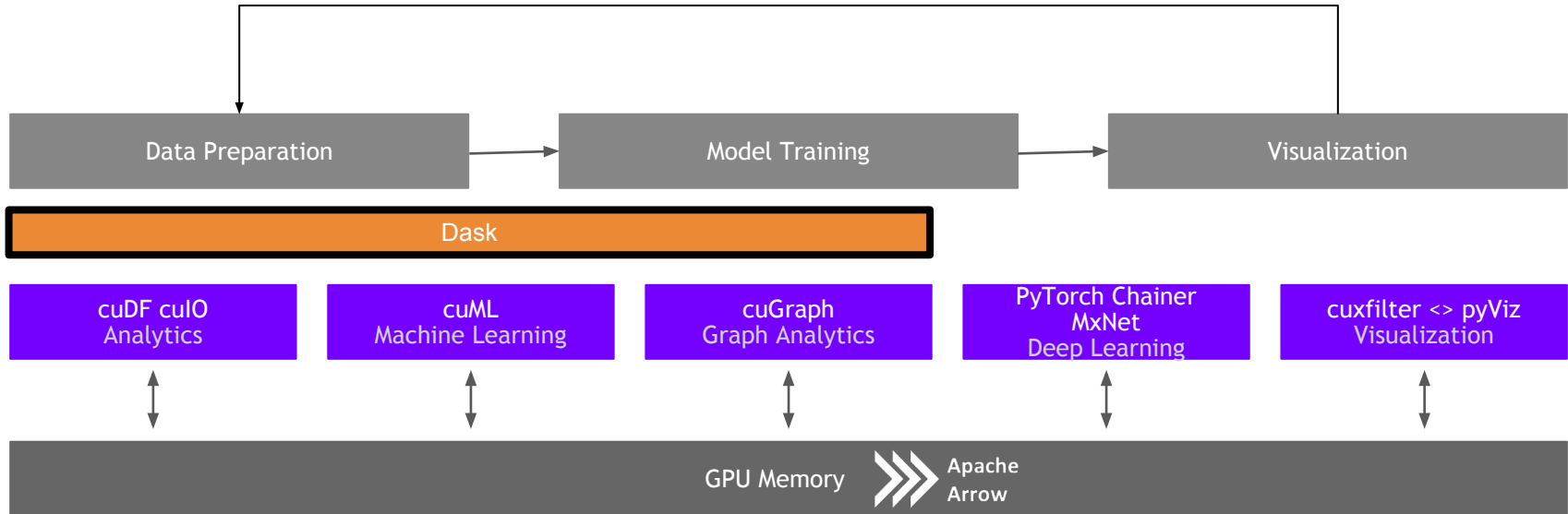
## End-to-End Accelerated GPU Data Science



Dask

# RAPIDS

## Scaling RAPIDS with Dask



# Why Dask?

## PyData Native

- **Easy Migration:** Built on top of NumPy, Pandas, Scikit-Learn, etc.
- **Easy Training:** With the same APIs
- **Trusted:** With the same developer community

## Deployable

- **HPC:** SLURM, PBS, LSF, SGE
- **Cloud:** Kubernetes
- **Hadoop/Spark:** Yarn



## Easy Scalability

- Easy to install and use on a laptop
- Scales out to thousand-node clusters

## Popular

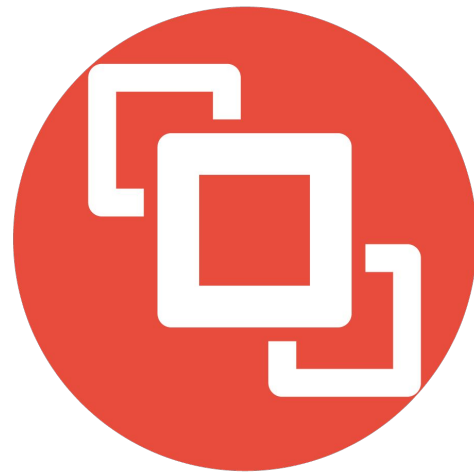
- Most common parallelism framework today in the PyData and SciPy community



# Why OpenUCX?

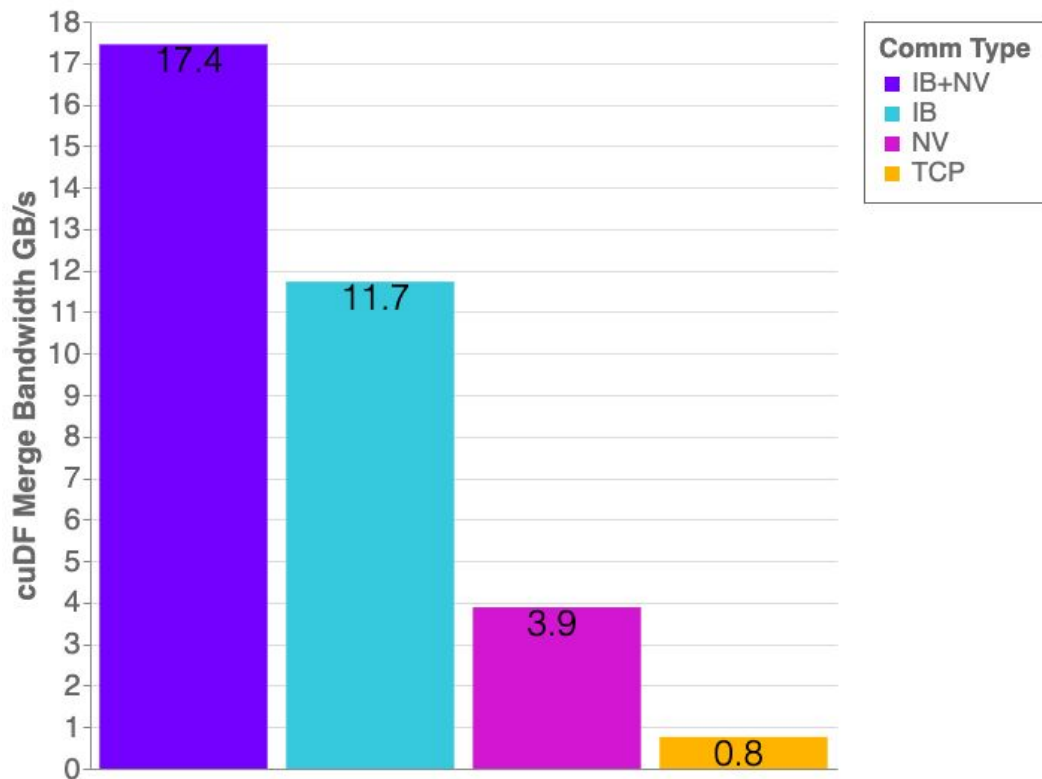
Bringing hardware accelerated communications to Dask

- TCP sockets are slow!
- UCX provides uniform access to transports (TCP, InfiniBand, shared memory, NVLink)
- Alpha Python bindings for UCX (ucx-py)
- Will provide best communication performance, to Dask based on available hardware on nodes/cluster



```
conda install -c conda-forge -c rapidsai \
  cudatoolkit=<CUDA version> ucx-proc=*=gpu ucx ucx-py
```

# Benchmarks: Distributed cuDF Random Merge



cuDF v0.13, UCX-PY 0.13

Running on NVIDIA DGX-1 (8GPUs):

GPU: NVIDIA Tesla V100 32GB

CPU: Intel(R) Xeon(R) CPU 8168

@ 2.70GHz

Benchmark Setup:

DataFrames: Left/Right 1x int64 column key  
column, 1x int64 value columns

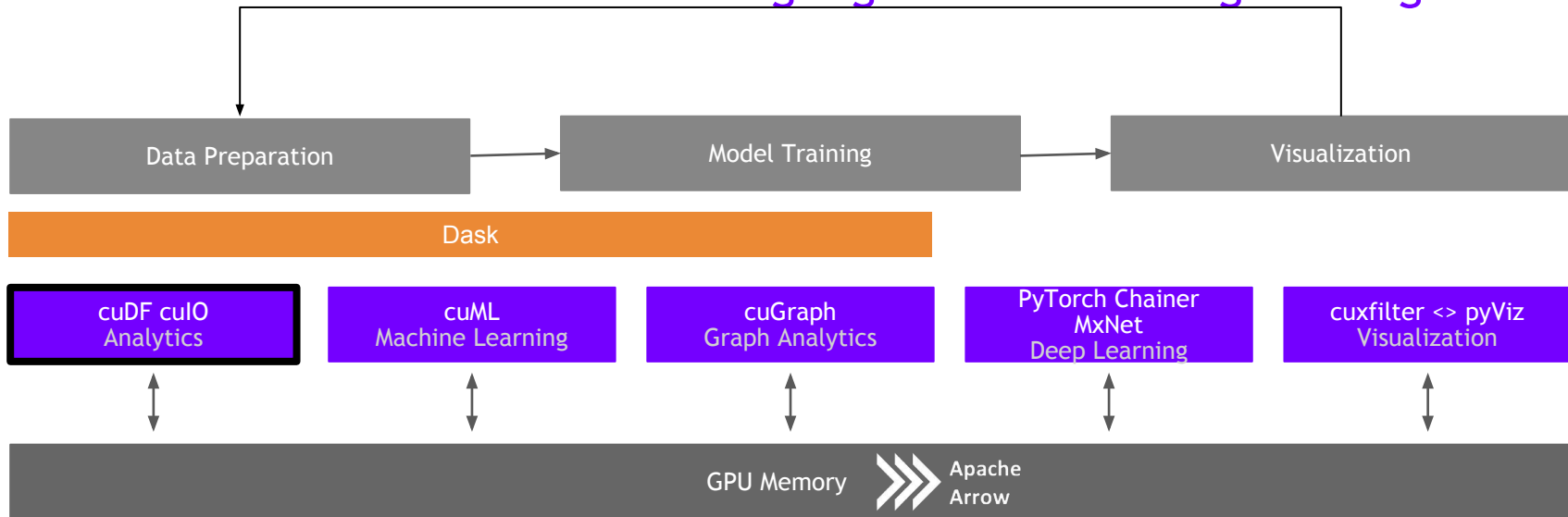
Merge: inner

30% of matching data balanced across each  
partition

cuDF

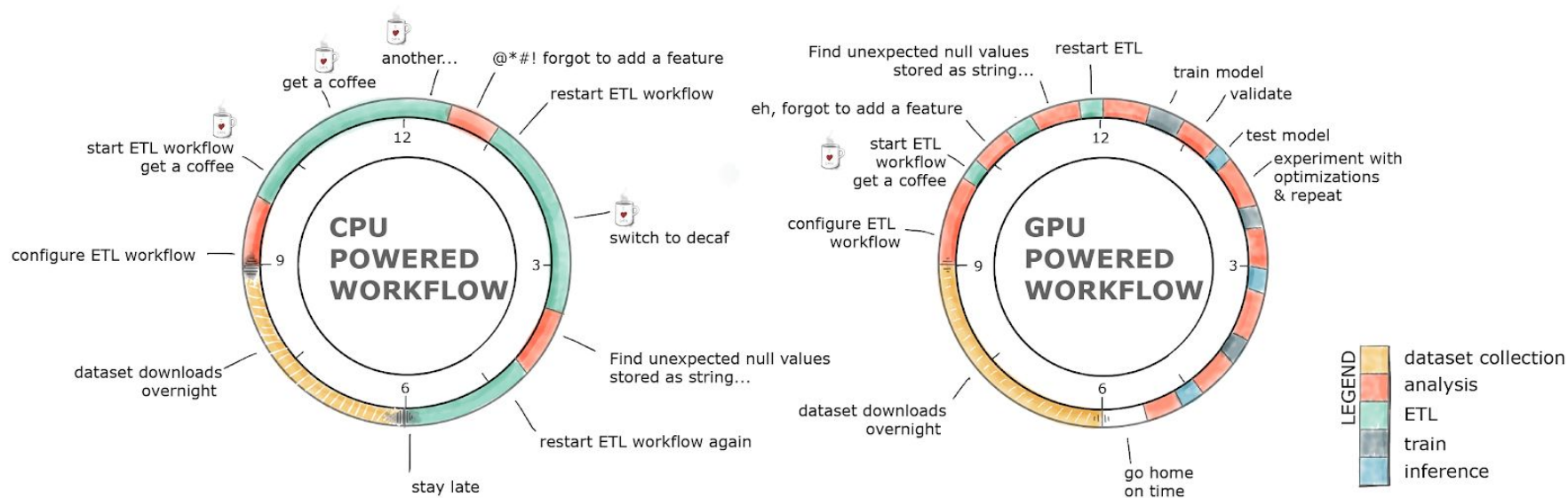
# RAPIDS

## GPU Accelerated data wrangling and feature engineering

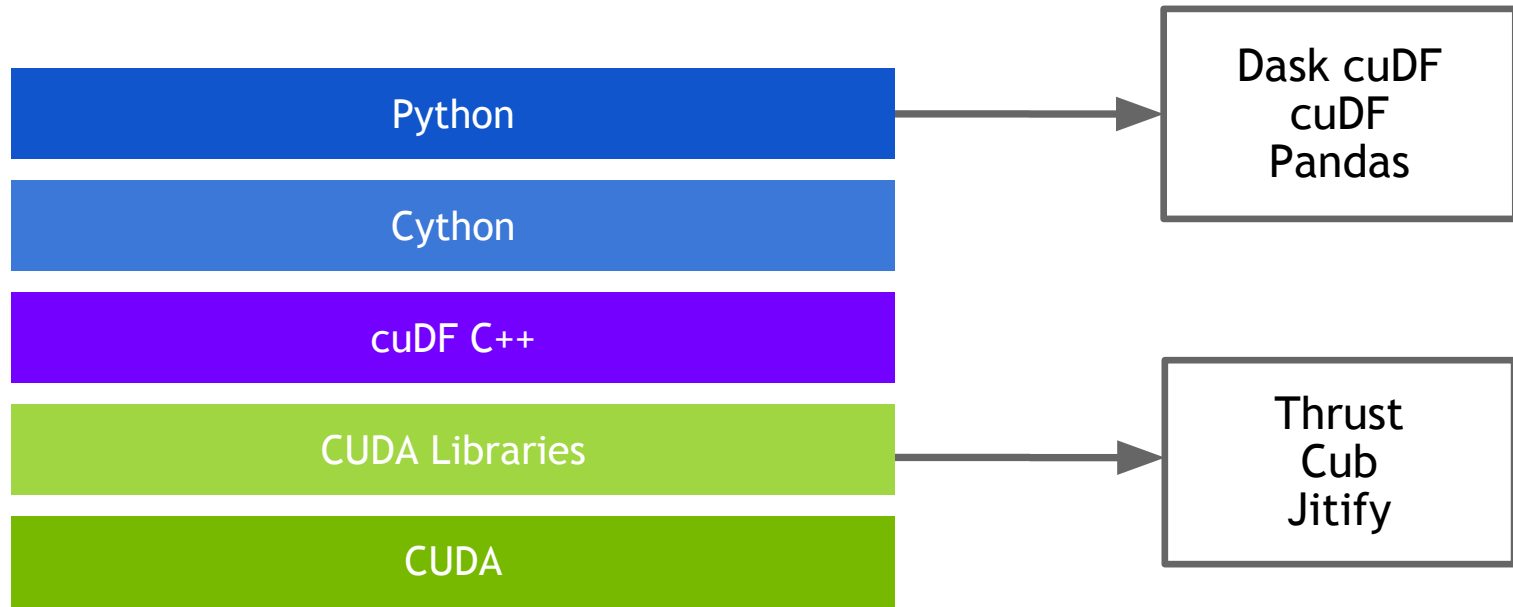


# GPU-Accelerated ETL

The average data scientist spends 90+% of their time in ETL as opposed to training models



# ETL Technology Stack



# ETL: the Backbone of Data Science

libcuDF is...

## CUDA C++ Library

- Table (dataframe) and column types and algorithms
- CUDA kernels for sorting, join, groupby, reductions, partitioning, elementwise operations, etc.
- Optimized GPU implementations for strings, timestamps, numeric types (more coming)
- Primitives for scalable distributed ETL

```
std::unique_ptr<table>
gather(table_view const& input,
      column_view const& gather_map, ...)
{
    // return a new table containing
    // rows from input indexed by
    // gather_map
}
```



# ETL: the Backbone of Data Science

## cuDF is...

### Python Library

- A Python library for manipulating GPU DataFrames following the Pandas API
- Python interface to CUDA C++ library with additional functionality
- Creating GPU DataFrames from Numpy arrays, Pandas DataFrames, and PyArrow Tables
- JIT compilation of User-Defined Functions (UDFs) using Numba

```
In [2]: #Read in the data. Notice how it decompresses as it reads the data into memory.
gdf = cudf.read_csv('/rapids/Data/black-friday.zip')
```

```
In [3]: #Taking a look at the data. We use "to_pandas()" to get the pretty printing.
gdf.head().to_pandas()
```

```
Out[3]:
```

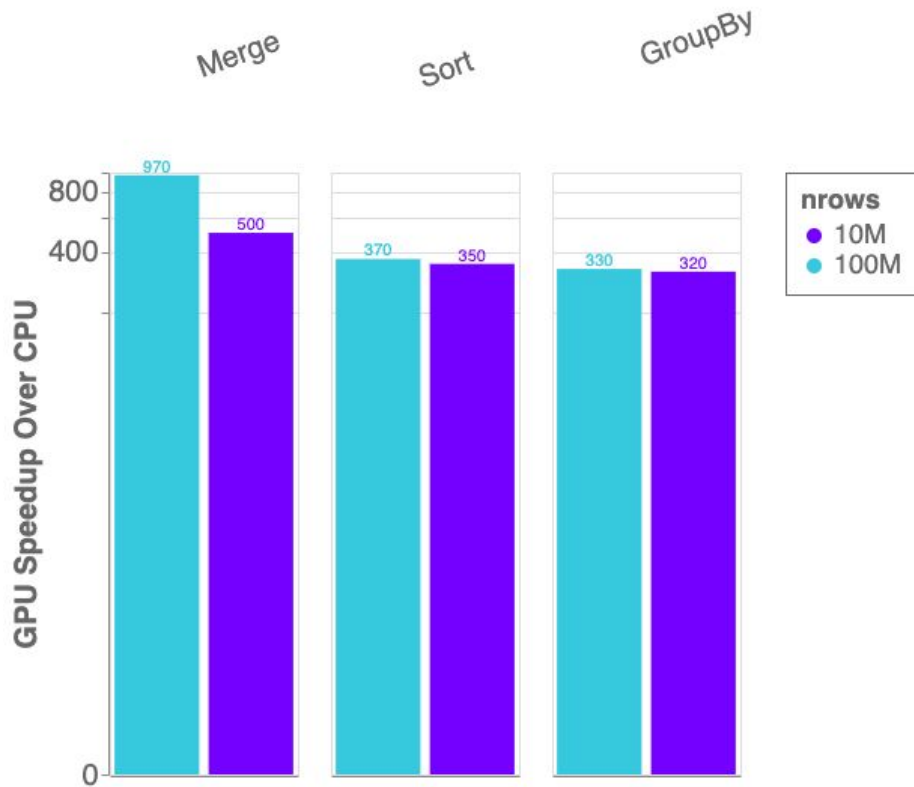
	User_ID	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Category
0	1000001	P00069042	F	0-17	10	A	2	0	3
1	1000001	P00248942	F	0-17	10	A	2	0	1
2	1000001	P00087842	F	0-17	10	A	2	0	12
3	1000001	P00085442	F	0-17	10	A	2	0	12
4	1000002	P00285442	M	55+	16	C	4+	0	8

```
In [6]: #grabbing the first character of the years in city string to get rid of plus sign, and converting to int
gdf['city_years'] = gdf.Stay_In_Current_City_Years.str.get(0).stoi()
```

```
In [7]: #Here we can see how we can control what the value of our dummies with the replace method and turn strings to ints
gdf['City_Category'] = gdf.City_Category.str.replace('A', '1')
gdf['City_Category'] = gdf.City_Category.str.replace('B', '2')
gdf['City_Category'] = gdf.City_Category.str.replace('C', '3')
gdf['City_Category'] = gdf['City_Category'].str.stoi()
```



# Benchmarks: single-GPU Speedup vs. Pandas



cuDF v0.13, Pandas 0.25.3

Running on NVIDIA DGX-1:

GPU: NVIDIA Tesla V100 32GB

CPU: Intel(R) Xeon(R) CPU E5-2698 v4  
@ 2.20GHz

Benchmark Setup:

RMM Pool Allocator Enabled

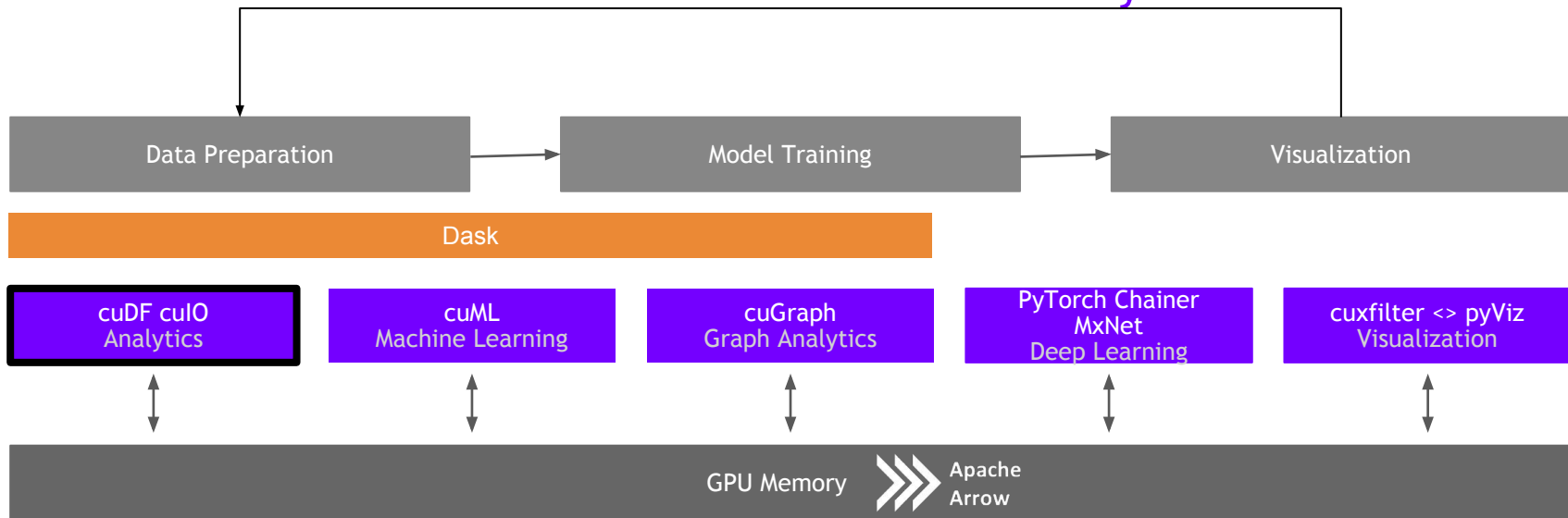
DataFrames: 2x int32 columns key columns,  
3x int32 value columns

Merge: inner

GroupBy: count, sum, min, max calculated  
for each value column

# ETL: the Backbone of Data Science

cuDF is not the end of the story



# ETL: the Backbone of Data Science

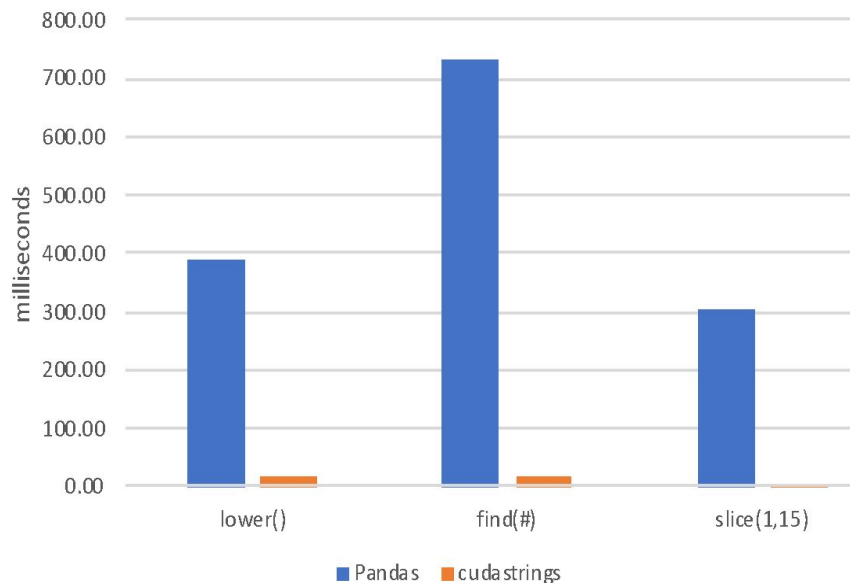
## String Support

### Current v0.13 String Support

- Regular Expressions
- Element-wise operations
  - Split, Find, Extract, Cat, Typecasting, etc...
- String GroupBys, Joins, Sorting, etc.
- Categorical columns fully on GPU
- Native String type in libcudf C++

### Future v0.14+ String Support

- Further performance optimization
- JIT-compiled String UDFs



# Extraction is the Cornerstone

## cuDF I/O for Faster Data Loading

- Follow Pandas APIs and provide >10x speedup
- CSV Reader - v0.2, CSV Writer v0.8
- Parquet Reader - v0.7, Parquet Writer v0.12
- ORC Reader - v0.7, ORC Writer v0.10
- JSON Reader - v0.8
- Avro Reader - v0.9
- GPU Direct Storage integration in progress for bypassing PCIe bottlenecks!
- Key is GPU-accelerating both parsing and decompression wherever possible

```
1]: import pandas, cudf

2]: %time len(pandas.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
CPU times: user 25.9 s, sys: 3.26 s, total: 29.2 s
Wall time: 29.2 s
2]: 12748986

3]: %time len(cudf.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
CPU times: user 1.59 s, sys: 372 ms, total: 1.96 s
Wall time: 2.12 s
3]: 12748986

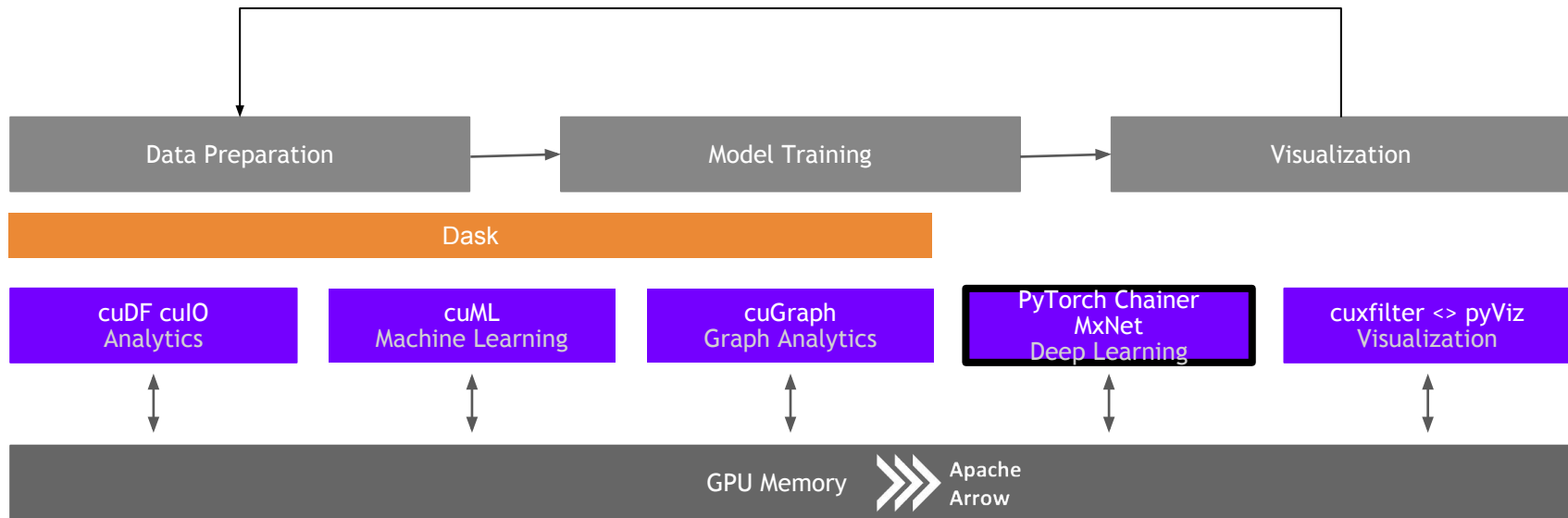
4]: !du -hs data/nyc/yellow_tripdata_2015-01.csv
1.9G    data/nyc/yellow_tripdata_2015-01.csv
```

Source: Apache Crail blog: [SQL Performance: Part 1 - Input File Formats](#)

ETL is not just DataFrames!

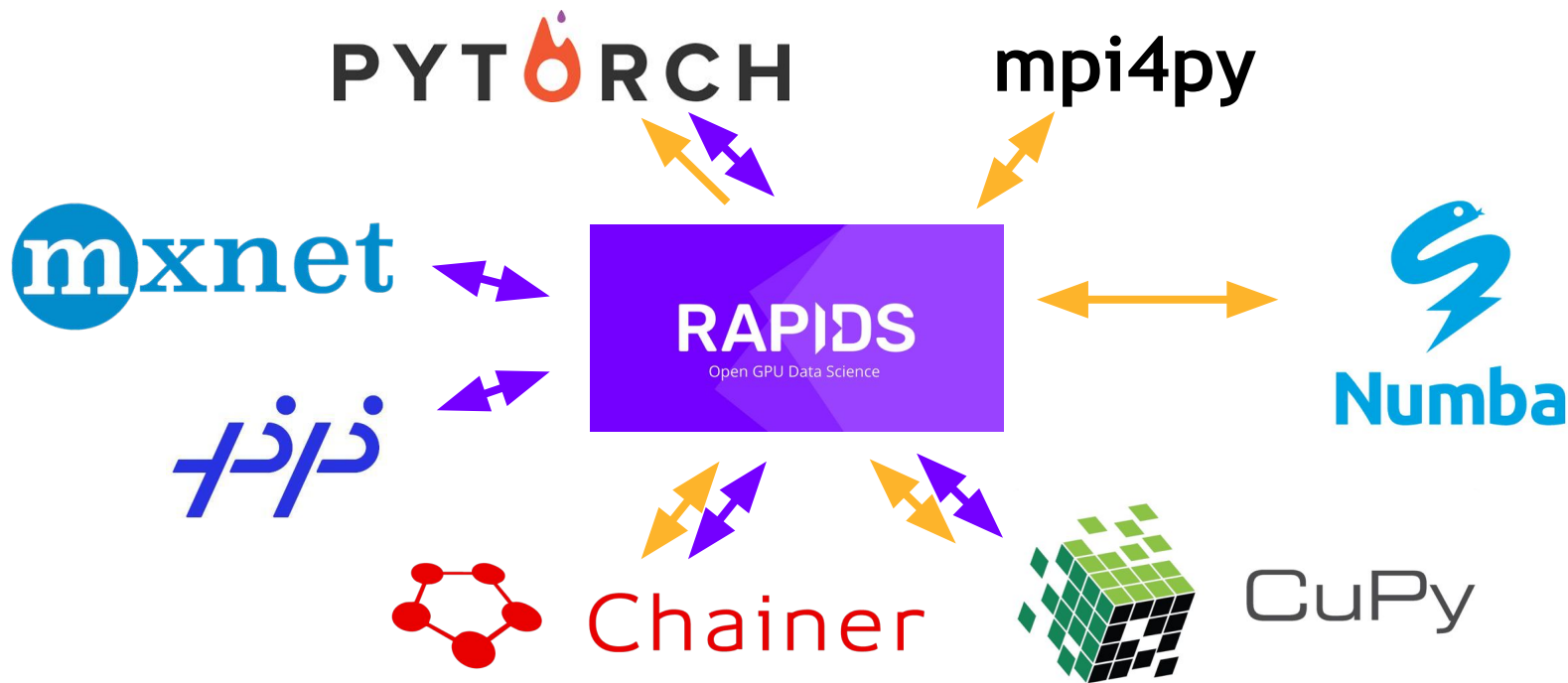
# RAPIDS

Building bridges into the array ecosystem



# Interoperability for the Win

DLPack and `__cuda_array_interface__`



# ETL: Arrays and DataFrames

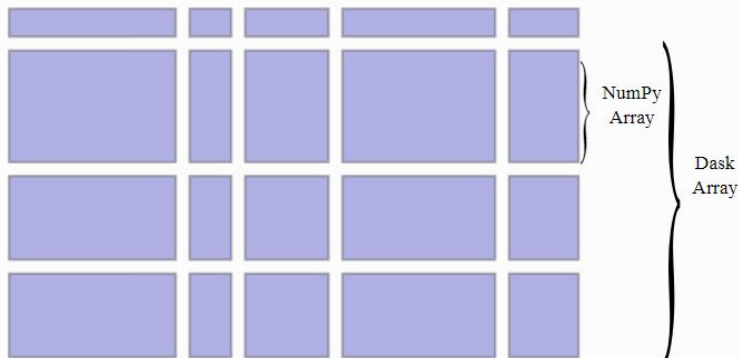
Dask and CUDA Python arrays



Chainer



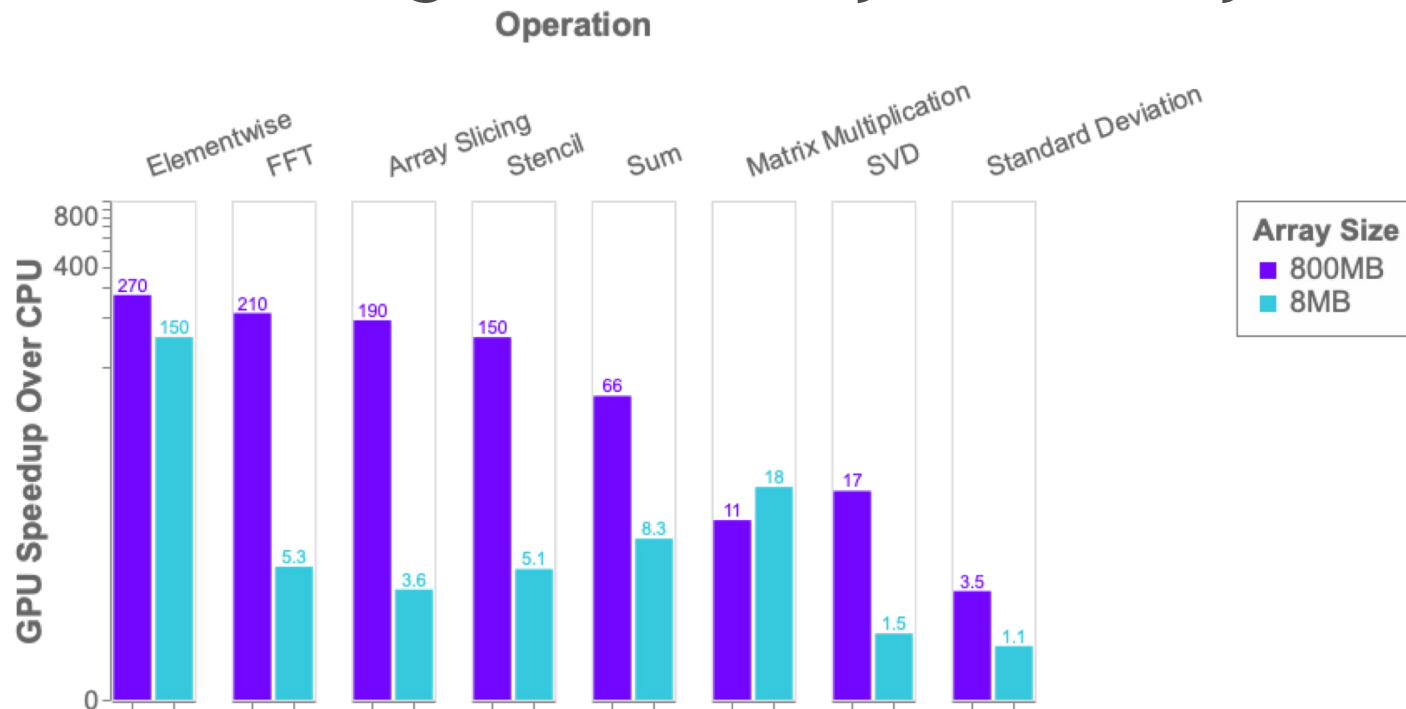
CuPy



- Scales NumPy to distributed clusters
- Used in climate science, imaging, HPC analysis up to 100TB size
- Now seamlessly accelerated with GPUs



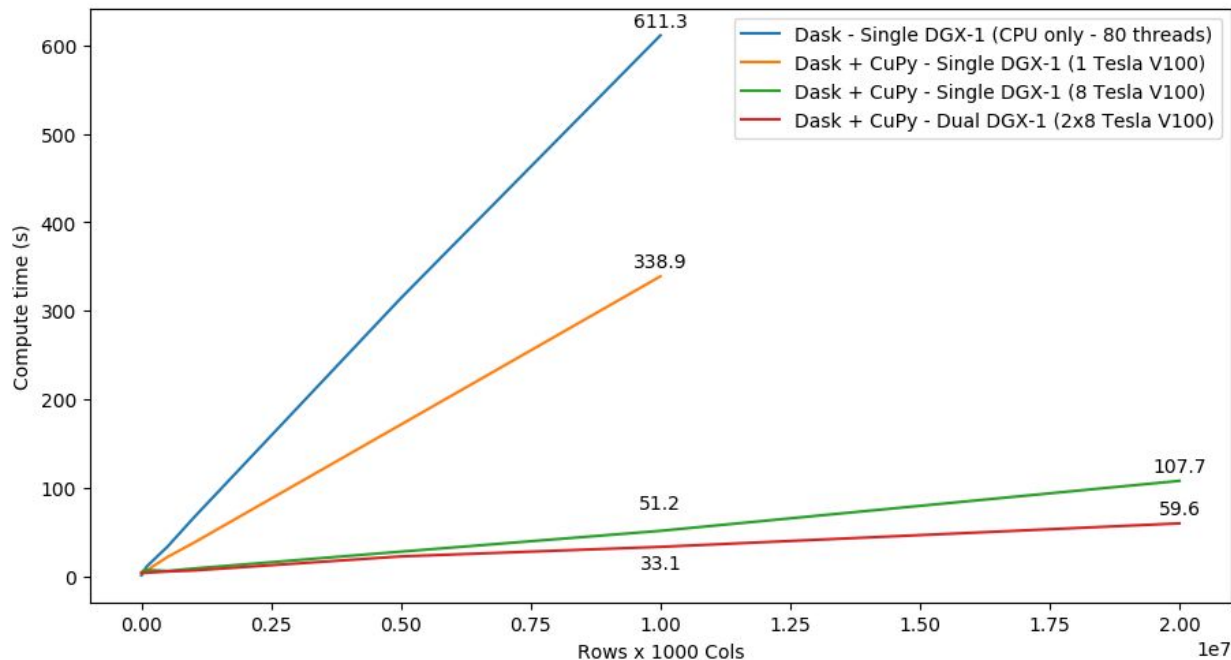
# Benchmark: single-GPU CuPy vs NumPy



More details: <https://blog.dask.org/2019/06/27/single-gpu-cupy-benchmarks>

# SVD Benchmark

## Dask and CuPy Doing Complex Workflows



# Petabyte Scale Analytics with Dask and CuPy

Architecture	Time
Single CPU Core	2hr 39min
Forty CPU Cores	11min 30s
One GPU	1min 37s
Eight GPUs	19s

<https://blog.dask.org/2019/01/03/dask-array-gpus-first-steps>



## 3.2 PETABYTES IN LESS THAN 1 HOUR

Distributed GPU array | parallel reduction | using 76x GPUs

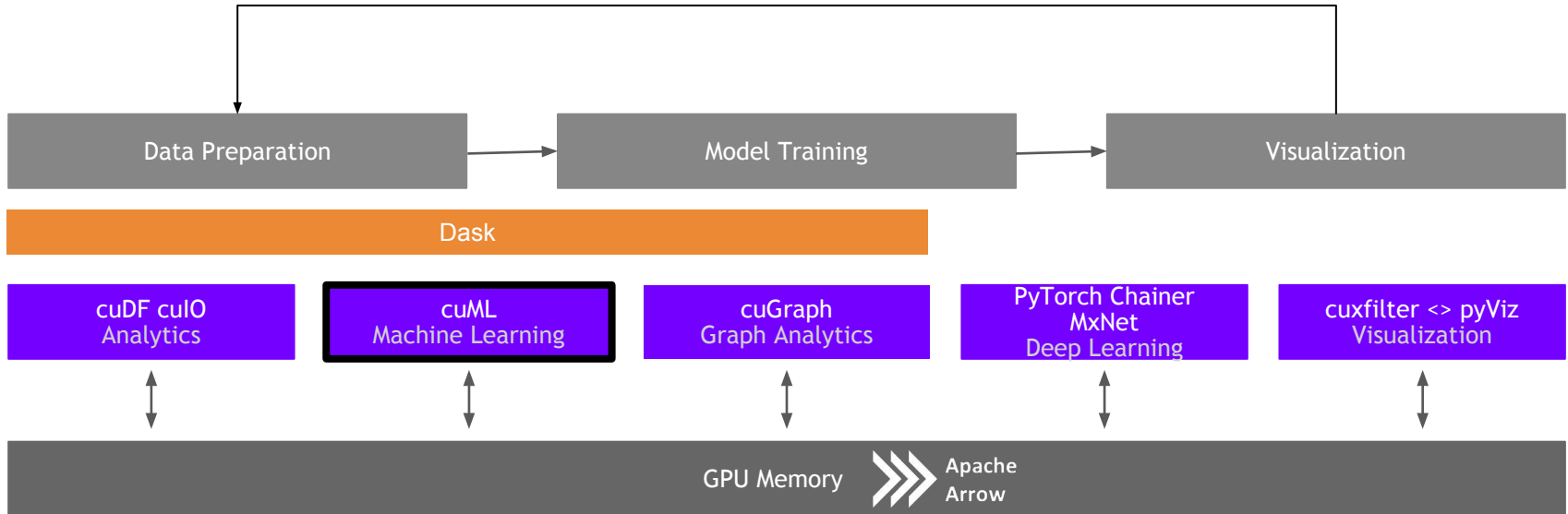
Array size	Wall Time (data creation + compute)
3.2 PB (20M x 20M doubles)	54 min 51 s

**Cluster configuration:** 20x GCP instances, each instance has:  
**CPU:** 1 VM socket (Intel Xeon CPU @ 2.30GHz), 2-core, 2 threads/core, 132GB mem, GbE ethernet, 950 GB disk  
**GPU:** 4x NVIDIA Tesla P100-16GB-PCIe (total GPU DRAM across nodes 1.22 TB)  
**Software:** Ubuntu 18.04, RAPIDS 0.5.1, Dask=1.1.1, Dask-Distributed=1.1.1, CuPY=5.2.0, CUDA 10.0.130

cuML

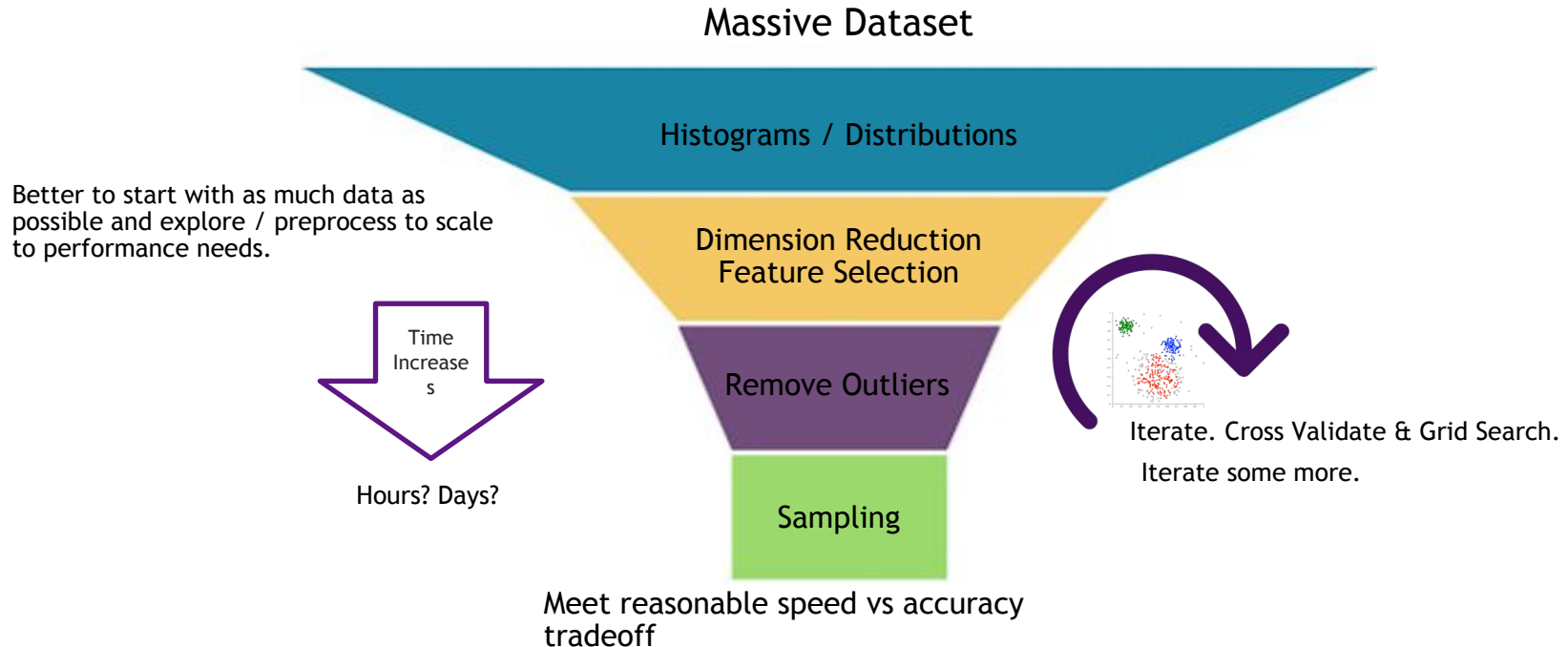
# Machine Learning

More models more problems

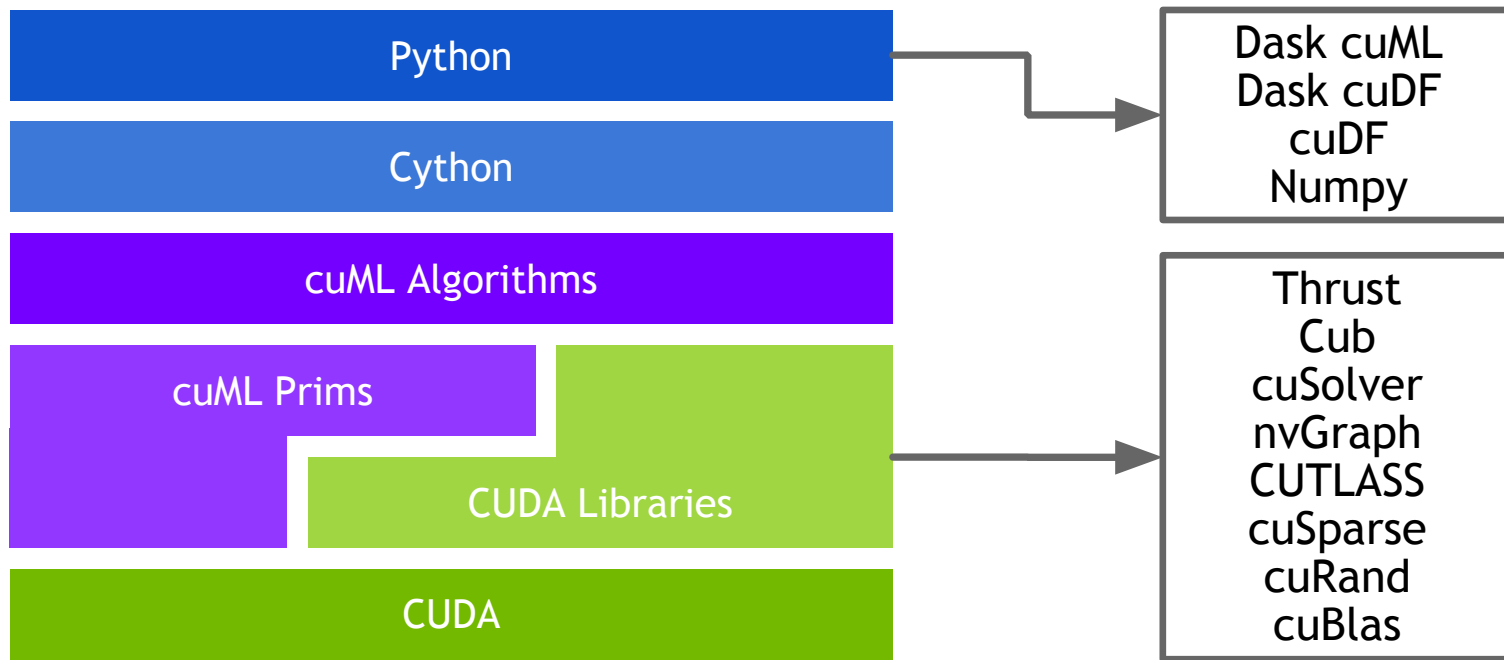


# Problem

Data sizes continue to grow

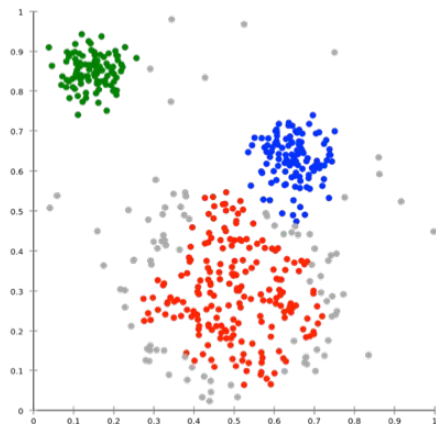


# ML Technology Stack



# Algorithms

## GPU-accelerated Scikit-Learn



Cross Validation

Hyper-parameter Tuning

More to  
come!

Classification / Regression

Inference

Clustering

Decomposition & Dimensionality Reduction

Time Series

Decision Trees / **Random Forests**  
**Linear Regression**  
Logistic Regression  
K-Nearest Neighbors  
Support Vector Machines

Random forest / GBDT inference

K-Means  
DBSCAN  
Spectral Clustering

Principal Components  
Singular Value Decomposition  
UMAP  
Spectral Embedding  
T-SNE

Holt-Winters  
**Seasonal ARIMA**

Key:

- Preexisting
- **NEW or enhanced for 0.13**



# RAPIDS matches common Python APIs

## CPU-Based Clustering

```
from sklearn.datasets import make_moons
import pandas
```

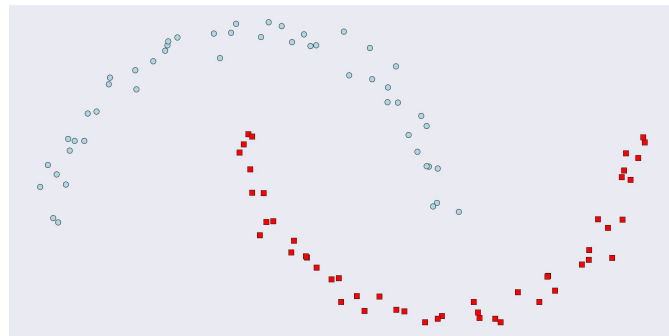
```
X, y = make_moons(n_samples=int(1e2),
                  noise=0.05, random_state=0)
```

```
X = pandas.DataFrame({'fea%d%i': X[:, i]
                      for i in range(X.shape[1])})
```

```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)
```

```
dbscan.fit(X)
```

```
y_hat = dbscan.predict(X)
```



# RAPIDS matches common Python APIs

## GPU-Accelerated Clustering

```
from sklearn.datasets import make_moons  
import cudf
```

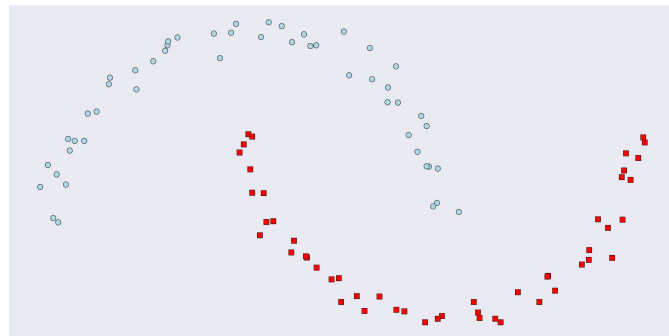
```
X, y = make_moons(n_samples=int(1e2),  
                  noise=0.05, random_state=0)
```

```
X = cudf.DataFrame({'fea%d'%i: X[:, i]  
                  for i in range(X.shape[1])})
```

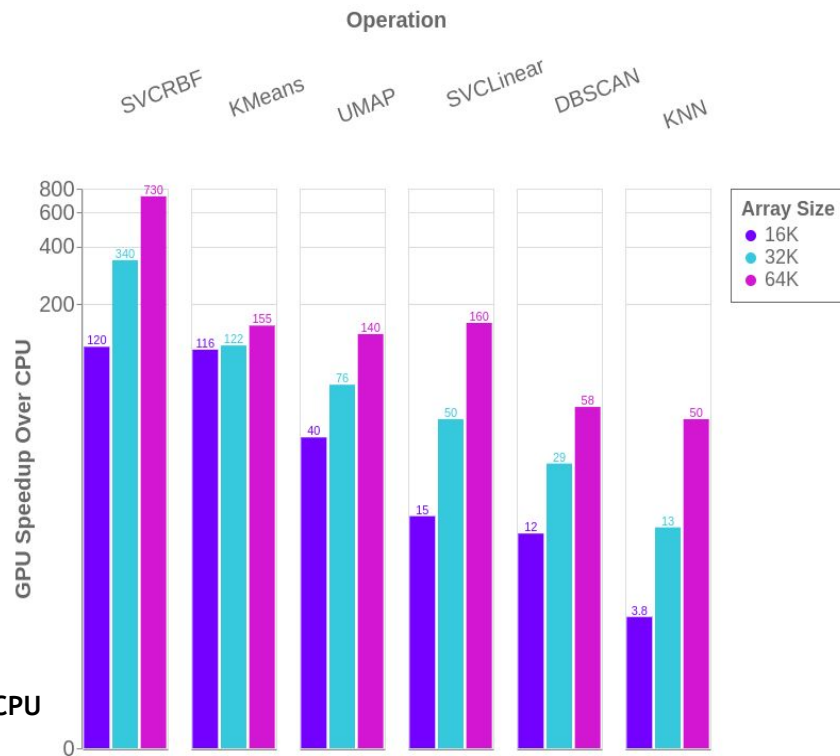
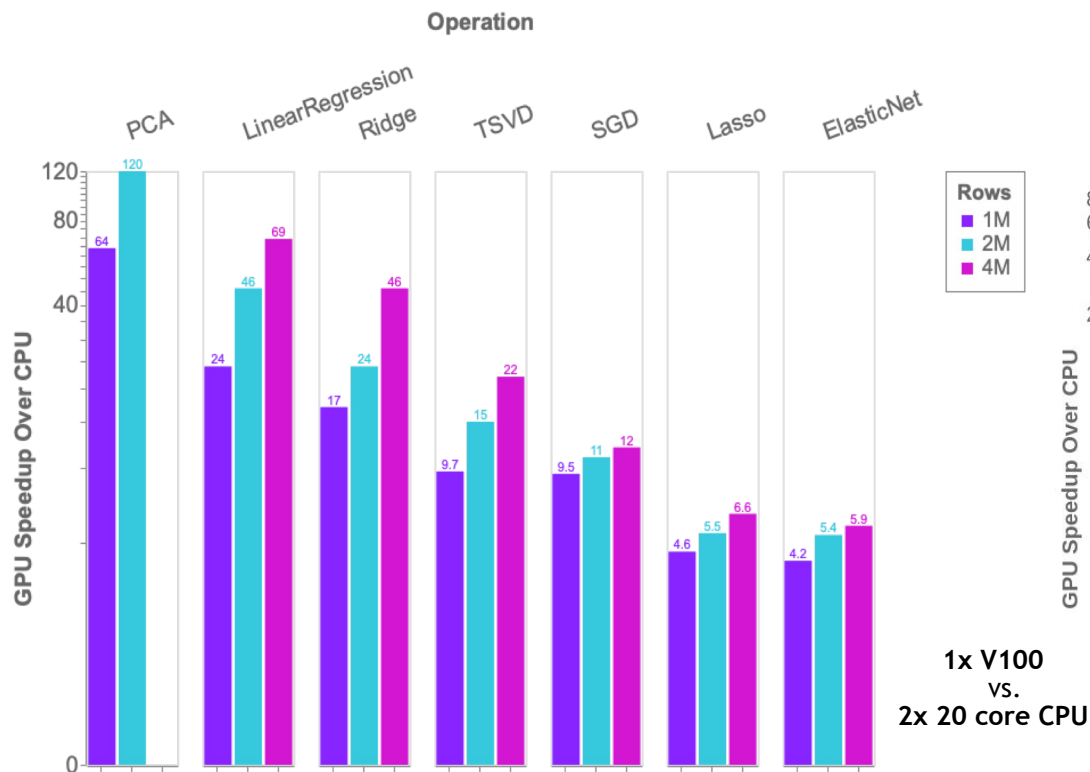
```
from cuml import DBSCAN  
dbscan = DBSCAN(eps = 0.3, min_samples = 5)
```

```
dbscan.fit(X)
```

```
y_hat = dbscan.predict(X)
```



# Benchmarks: single-GPU cuML vs scikit-learn

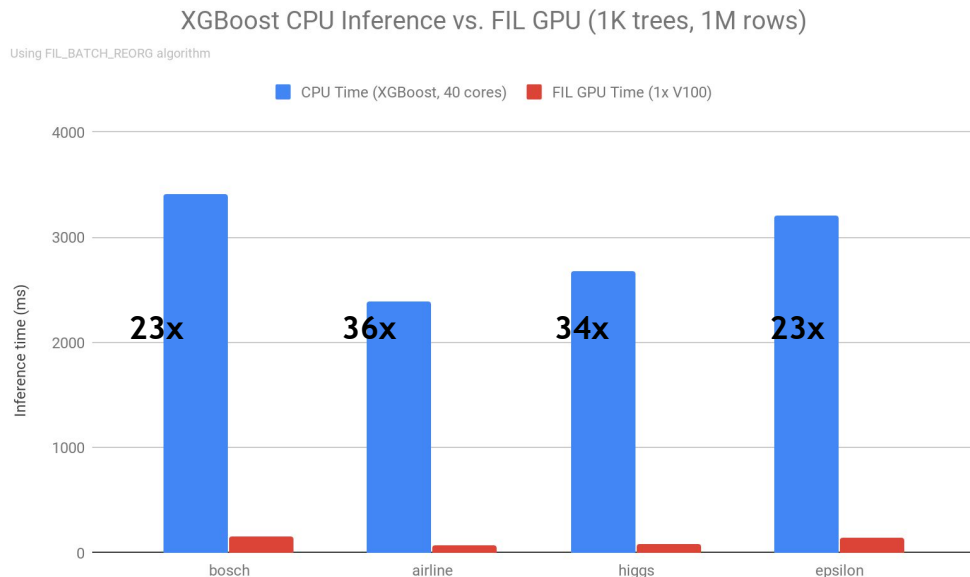


# Forest Inference

## Taking models from training to production

cuML's Forest Inference Library accelerates prediction (inference) for random forests and boosted decision trees:

- Works with existing saved models (XGBoost, LightGBM, scikit-learn RF cuML RF soon)
- Lightweight Python API
- Single V100 GPU can infer up to 34x faster than XGBoost dual-CPU node
- Over 100 million forest inferences per sec (with 1000 trees) on a DGX-1 for large (sparse) or dense models

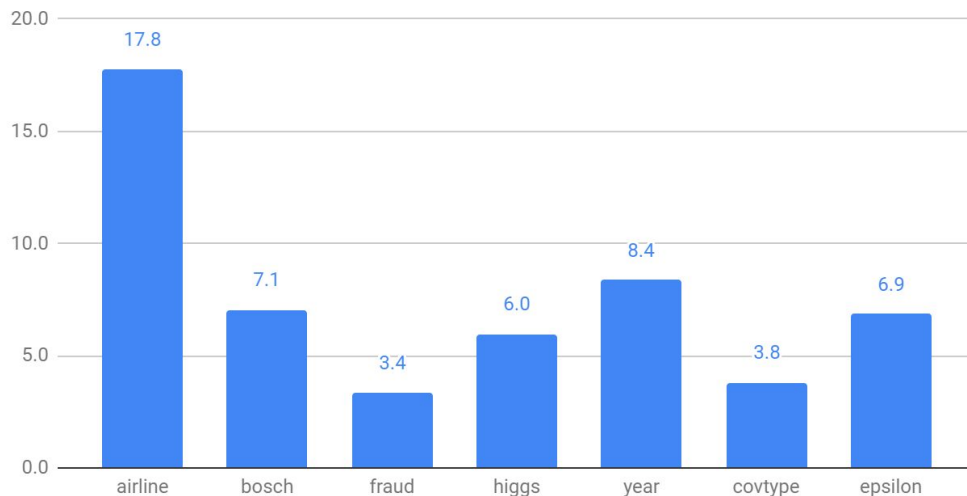




+ *dmlc*  
***XGBoost***

- RAPIDS works closely with the XGBoost community to accelerate GBDTs on GPU
- The default rapids conda metapackage includes XGBoost
- XGBoost can seamlessly load data from cuDF dataframes and cuPy arrays
- Dask allows XGBoost to scale to arbitrary numbers of GPUs
- With the *gpu\_hist* tree method, a single GPU can outpace 10s to 100s of CPUs
- Version 1.0 of XGBoost launched with the RAPIDS 0.13 stack.

XGBoost GPU Speedup - Single V100 vs Dual 20-core Xeon E5-2698



# Road to 1.0

## March 2020 - RAPIDS 0.13

cuML	Single-GPU	Multi-GPU	Multi-Node-Multi-GPU
Gradient Boosted Decision Trees (GBDT)			
Linear Regression			
Logistic Regression			
Random Forest			
K-Means			
K-NN			
DBSCAN			
UMAP			
Holt-Winters			
ARIMA			
t-SNE			
Principal Components			
Singular Value Decomposition			
SVM			

# Road to 1.0

## 2020 - RAPIDS 1.0

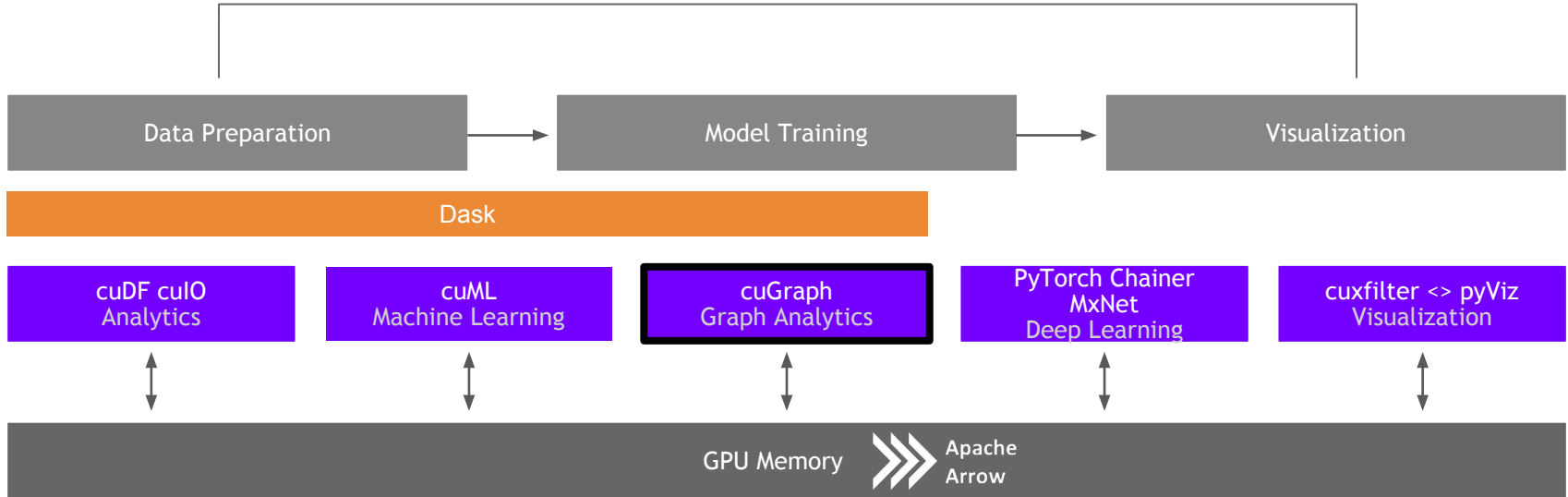
cuML	Single-GPU	Multi-Node-Multi-GPU
Gradient Boosted Decision Trees (GBDT)		
Linear Regression (regularized)		
Logistic Regression		
Random Forest		
K-Means		
K-NN		
DBSCAN		
UMAP		
Holt-Winters		
ARIMA		
t-SNE		
Principal Components		
Singular Value Decomposition		
SVM		

cuGraph



# Graph Analytics

More connections more insights



# Goals and Benefits of cuGraph

Focus on Features and User Experience

## Breakthrough Performance

- Up to 500 million edges on a single 32GB GPU
- Multi-GPU support for scaling into the billions of edges

## Multiple APIs

- **Python:** Familiar NetworkX-like API
- **C/C++:** lower-level granular control for application developers

## Seamless Integration with cuDF and cuML

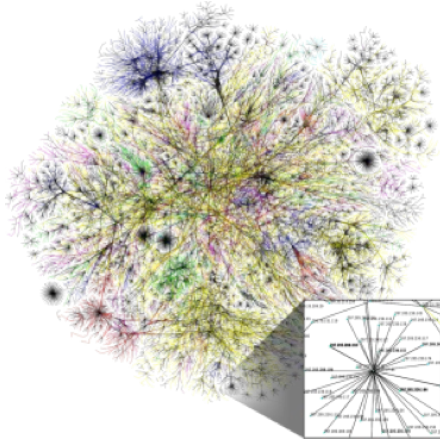
- Property Graph support via DataFrames

## Growing Functionality

- Extensive collection of algorithm, primitive, and utility functions

# Algorithms

## GPU-accelerated NetworkX



Graph Classes

Structure

Multi-GPU

Utilities

More to  
come!

Community

Components

Link Analysis

Link Prediction

Traversal

Centrality

Spectral Clustering  
Balanced-Cut  
Modularity Maximization

Louvain  
Ensemble Clustering for Graphs  
Subgraph Extraction  
KCore and KCore Number  
Triangle Counting  
***K-Truss***

Weakly Connected Components  
Strongly Connected Components

Page Rank (Multi-GPU)  
Personal Page Rank

Jaccard  
Weighted Jaccard  
Overlap Coefficient

Single Source Shortest Path (SSSP)  
Breadth First Search (BFS)

Katz  
***Betweenness Centrality***

# Multi-GPU PageRank Performance

PageRank portion of the HiBench benchmark suite

HiBench Scale	Vertices	Edges	CSV File (GB)	# of V100 GPUs	# of CPU Threads	PageRank for 3 Iterations (secs)
Huge	5,000,000	198,000,000	3	1		1.1
BigData	50,000,000	1,980,000,000	34	3		5.1
BigData x2	100,000,000	4,000,000,000	69	6		9.0
BigData x4	200,000,000	8,000,000,000	146	12		18.2
BigData x8	400,000,000	16,000,000,000	300	16		31.8
BigData x8	400,000,000	16,000,000,000	300		800*	5760*

\*BigData x8, 100x 8-vCPU nodes, Apache Spark GraphX  $\Rightarrow$  96 mins!

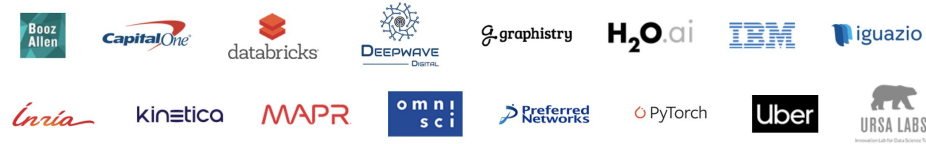
Community

# Ecosystem Partners

## CONTRIBUTORS



## ADOPTERS

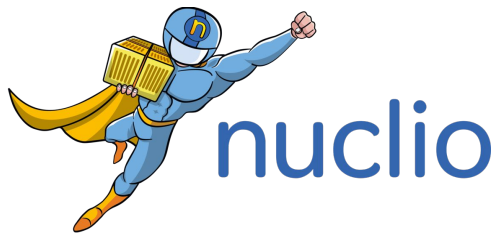


## OPEN SOURCE



# Building on top of RAPIDS

A bigger, better, stronger ecosystem for all



**High-Performance  
Serverless event and  
data processing that  
utilizes RAPIDS for GPU  
Acceleration**



**GPU accelerated SQL  
engine built on top of  
RAPIDS**

## Streamz

**Distributed stream  
processing using  
RAPIDS and Dask**

# Easy Installation

## Interactive Installation Guide

### RAPIDS RELEASE SELECTOR

RAPIDS is available as conda packages, docker images, and from source builds. Use the tool below to select your preferred method, packages, and environment to install RAPIDS. Certain combinations may not be possible and are dimmed automatically. Be sure you've met the required [prerequisites above](#) and see the [details below](#).

	↓ Preferred ↓		↓ Advanced ↓				
METHOD	Conda 🐍	Docker + Examples 🐳	Docker + Dev Env 🐳	Source ⚙️			
RELEASE	Stable (0.13)		Nightly (0.14a)				
PACKAGES	All Packages	cuDF	cuML	cuGraph	cuSignal	cuSpatial	cuxfilter
LINUX	Ubuntu 16.04 🐧	Ubuntu 18.04 🐧	CentOS 7 🐧	RHEL 7 🐧			
PYTHON	Python 3.6		Python 3.7				
CUDA	CUDA 10.0		CUDA 10.1.2		CUDA 10.2		

📌 NOTE: Ubuntu 16.04/18.04 & CentOS 7 use the same `conda install` commands.

COMMAND

```
conda install -c rapidsai -c nvidia -c conda-forge \
  -c defaults rapids=0.13 python=3.6
```

COPY COMMAND 📄

DETAILS BELOW





# Contribute Back

Issues, feature requests, PRs, Blogs, Tutorials, Videos, QA...bring your best!

The image shows a vertical stack of four GitHub repository pages for RAPIDS projects. Each page includes the repository name, a brief description, tags for languages and frameworks, and statistics like stars, forks, and open issues. A green line graph on the right of each repository indicates recent activity.

- cuml**: cuML - RAPIDS Machine Learning Library. Tags: machine-learning, gpu, machine-learning-algorithms, cuda, nvidia. Stats: 608 stars, 186 issues need help. Updated 9 minutes ago.
- cudf**: cuDF - GPU DataFrame Library. Tags: anaconda, gpu, arrow, machine-learning-algorithms, h2o, cuda, pandas. Stats: 1,699 stars, 325 issues need help. Updated 31 minutes ago.
- notebooks-contrib**: RAPIDS Community Notebooks. Tags: Jupyter Notebook, Apache-2.0. Stats: 70 stars, 10 issues need help. Updated 40 minutes ago.
- cugraph**: cuGraph. Tags: Cuda, Apache-2.0. Stats: 172 stars, 58 issues need help. Updated 3 minutes ago.

The image shows a blog post from Walmart Labs. The title is "How GPU Computing literally saved me at work?". The subtitle is "Python+GPU = Power, 2 Days to 20 seconds". The author is Abhishek Mungoli, and the post was updated 9 minutes ago. The blog header includes links for TECH BLOG, ENGINEERING, DATA SCIENCE, INFOSEC, UX DESIGN, LEADERSHIP, and ABOUT.

The image shows a tweet from John Murray (@MurrayData). The tweet compares CPU and GPU performance for a project. The text reads: "Comparison CPU vs GPU @rapidsai to project 100 million x,y points to lat/lon to 0.01mm accuracy. CPU 1 core c 65 mins, multicore c 13 mins, GPU #RAPIDSAI 2 seconds. I optimised the code since previous run. Dell T7910 Xeon E5-2640V4x2/NVIDIA Titan Xp cc @NvidiaAI @marc\_stampfli". Below the text is a terminal screenshot showing the execution of a Python script, comparing CPU and GPU performance.

```
john@plato:~/Source/Python/misc$ python crs_test.py
Generating Data
CPU Iterative
4005.0377202 seconds
CPU mapped
3957.19386101 seconds
CPU multiprocessing
788.550751209 seconds
GPU Rapids
2.103230476 seconds
```

## Getting Started with cuDF (RAPIDS)

The image shows a tweet from Darren Ramsook. The tweet is titled "Getting Started with cuDF (RAPIDS)". The author is Darren Ramsook, and the tweet was updated 3 minutes ago. The tweet includes a "Follow" button.

# Getting Started

# RAPIDS Docs

## Improved and easier to use!

The screenshot shows a web browser window with the URL `docs.rapids.ai/api/cudf/stable/10min.html`. The page layout includes a left sidebar with navigation links: Home, cudf, stable (0.13), a search bar, and a 'CONTENTS' section. The 'CONTENTS' section lists: API Reference, 10 Minutes to cuDF and Dask-cuDF (which is expanded to show sub-items like 'What are these Libraries?', 'When to use cuDF and Dask-cuDF', 'Persisting Data', 'Wait', 'Multi-GPU with Dask-cuDF', '10 Minutes to Dask-XGBoost', '10 Minutes to cuDF and CuPy', 'Overview of User Defined Functions with cuDF', and 'Developer Documentation'), and Developer Documentation. The main content area displays the title '10 Minutes to cuDF and Dask-cuDF' with a 'View page source' link. Below the title is a paragraph stating it is modeled after '10 Minutes to Pandas'. A section titled 'What are these Libraries?' follows, describing cuDF and Dask. Another section, 'When to use cuDF and Dask-cuDF', provides guidance on when to use each library. At the bottom, a code block shows a Jupyter Notebook cell with Python code for importing libraries and seeding a random number generator.

docs.rapids.ai/api/cudf/stable/10min.html

Home  
cudf  
stable (0.13)  
Search docs

CONTENTS:  
API Reference  
10 Minutes to cuDF and Dask-cuDF  
What are these Libraries?  
When to use cuDF and Dask-cuDF  
Persisting Data  
Wait  
Multi-GPU with Dask-cuDF  
10 Minutes to Dask-XGBoost  
10 Minutes to cuDF and CuPy  
Overview of User Defined Functions with cuDF  
Developer Documentation

Docs » 10 Minutes to cuDF and Dask-cuDF [View page source](#)

## 10 Minutes to cuDF and Dask-cuDF

Modeled after 10 Minutes to Pandas, this is a short introduction to cuDF and Dask-cuDF, geared mainly for new users.

### What are these Libraries?

cuDF is a Python GPU DataFrame library (built on the Apache Arrow columnar memory format) for loading, joining, aggregating, filtering, and otherwise manipulating tabular data using a DataFrame style API.

Dask is a flexible library for parallel computing in Python that makes scaling out your workflow smooth and simple. On the CPU, Dask uses Pandas to execute operations in parallel on DataFrame partitions.

Dask-cuDF extends Dask where necessary to allow its DataFrame partitions to be processed by cuDF GPU DataFrames as opposed to Pandas DataFrames. For instance, when you call `dask_cudf.read_csv(...)`, your cluster's GPUs do the work of parsing the CSV file(s) with underlying `cudf.read_csv()`.

### When to use cuDF and Dask-cuDF

If your workflow is fast enough on a single GPU or your data comfortably fits in memory on a single GPU, you would want to use cuDF. If you want to distribute your workflow across multiple GPUs, have more data than you can fit in memory on a single GPU, or want to analyze data spread across many files at once, you would want to use Dask-cuDF.

```
[1]: import os

import numpy as np
import pandas as pd
import cudf
import dask_cudf

np.random.seed(12)

#### Portions of this were borrowed and adapted from the
#### cuDF cheatsheet, existing cuDF documentation,
#### and 10 Minutes to Pandas.
```

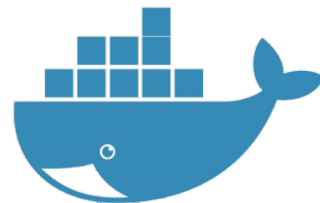
<https://docs.rapids.ai>

# RAPIDS

How do I get the software?



- <https://github.com/rapidsai>
- <https://anaconda.org/rapidsai/>



- <https://ngc.nvidia.com/registry/nvidia-rapidsai-rapidsai>
- <https://hub.docker.com/r/rapidsai/rapidsai/>

# THANK YOU

Nick Becker

[nicholas@nvidia.com](mailto:nicholas@nvidia.com)

# RAPIDS